Formation Entity Framework Core

Présentation

Module 0

Gaëtan GOUHIER

- Ingénieur en informatique
- Spécialisé dans le développement .NET
- Certifié Microsoft (70-483, 70-486, AZ-203) / Azure Developer Associate
- Développeur et formateur indépendant depuis 2016
- https://www.gaetan-gouhier.net/

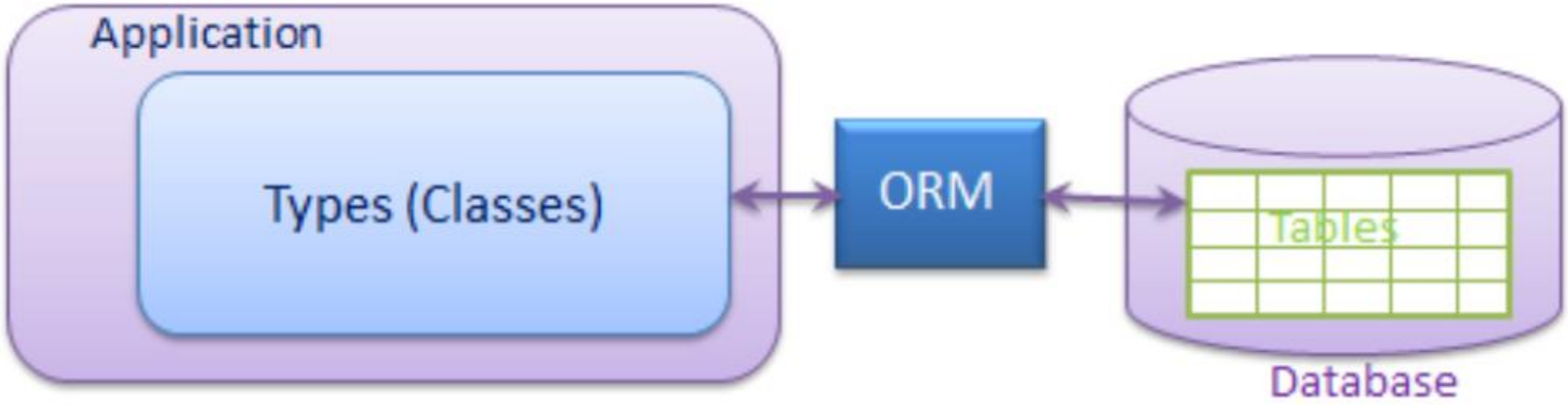
Sommaire

Introduction

Module 1

Introduction

- Le Framework Entity (désigné à travers l'acronyme EF) est un ORM (Object Relational Mapping) qui permet aux applications d’interagir avec des bases de données relationnelles.
- L'objectif est de fournir une couche d'abstraction nécessaire aux développeurs pour qu'ils n'accèdent plus directement à la base de données, mais par l'intermédiaire d'entités définies par un modèle appelé EDM (Entity Data Model).



EF Core

- Depuis 2016 Microsoft a lancé sa nouvelle version core
- C'est une version d'EF totalement réécrite
- Toutes les fonctionnalités d'EF 6 ne seront pas supportées
- Basé sur Code First pour être plus efficace, puissant, flexible et extensible
- Fonctionne sur Windows, Linux et OSX.
- Disponible sous licence Apache License v2 et disponible entièrement en Open Source sur GitHub
- Actuellement en version 5

EF Core VS EF 6

- Entity Framework Core est la nouvelle version
- Il est conseillé de partir sur EF Core sauf si
  - On a un projet existant en .NET Framework
  - On a des dépendances obligatoires à .NET Framework
- Le fonctionnement en DB First de EF 6 à EF Core a été grandement modifié
- Le fonctionnement en Code First de EF 6 à EF Core n'a pas beaucoup changé
- Le fait de passer d'EF 6.0 vers EF Core est une migration, pas une mise à jour.

Différents modes

Module 2

Mise en œuvre de Entity Framework

- La mise en œuvre de EF peut s'effectuer de deux façons :
  - DB First : à partir d'une base existante. Les entités sont générées à partir des tables
  - Code First : Les tables sont générées à partir des classes métiers
- Pour utiliser EF il va falloir rajouter des packages Nuget
  - Microsoft.EntityFrameworkCore
  - Microsoft.EntityFrameworkCore.Tools - pour les outils en ligne de commande
  - Microsoft.EntityFrameworkCore.SqlServer - pour base de donnée SQL Server

DB First

- Souvent utilisé quand
  - Base de données existante
  - DBA dédié
  - Besoins spécifiques
- Permet de générer nos classes à partir de la base de données
- Les classes générées sont des classes partielles, on pourra donc ajouter des méthodes ou des propriétés dans un autre fichier avec le même nom de classe, on ne modifie jamais un fichier généré

DB First

- Pour faire du Db First on va utiliser les lignes de commandes dans la ligne de commande NuGet (Package Manager Console)
- Deux types de commandes sont possibles
  - Soit en passant par la CLI
    - dotnet tool update --global dotnet-ef (pour installer de facon globale la CLI EF, à ne faire qu'une fois)
    - dotnet ef dbcontext scaffold ...
  - Soit en passant par le package Tools (les commandes ressembleront plus à EF6)
    - Scaffold-DbContext ...

DB First

- Pour lancer la génération des classes il va falloir passer par la commande Scaffold-DbContext
  - Scaffold-DbContext Microsoft.EntityFrameworkCore.SqlServer --help
- Microsoft.EntityFrameworkCore.SqlServer défini simplement quel provider utiliser
- Plusieurs options sont importantes
  - -d pour activer les Data Annotations
  - -c pour définir le nom du contexte généré
  - --context-dir pour définir le dossier du contexte
  - -f pour forcer l'écrasement des fichiers existants
  - -o pour définir le dossier des entités générées
- De plus il faudra définir la chaine de connexion à la BDD, notre commande pourra donc être
  - Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Location;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -d -f -o Models

Code First

- Souvent utilisé quand
  - Base de données non existante
  - Ensemble de développeur qui ne veulent / peuvent pas gérer la BDD
- Permet de générer la base de données à partir de nos classes

Les objets persistants

- Un objet persistant doit donc être une instance d'une classe C#. Cette classe doit respecter les standards
  - fournir des propriétés d'accès (get ; set ;)
  - fournir un constructeur par défaut.
  - doit être définie dans un namespace.
- Les types utilisables pour les propriétés sont :
  - les types primitifs du langage
  - les classes String et DateTime
  - Les autres classes qui génèreront des relations

Clé primaire

- Pour assurer l'unicité d'un enregistrement dans une table, il faut définir une clé primaire.
- La bonne pratique consiste à définir une clé primaire artificielle générée ce qui rend les aspects persistants plus indépendants des aspects métiers.
- Cette clé primaire doit se retrouver en tant qu'attribut de la classe, elle n'a aucun sens métier.
- La clé primaire doit respecter des conventions de nommage :
  - se nommer Id
  - avoir le nom de votre classe avec le suffixe Id.

```
public class Client
{
        public int Id { get; set; }
        // OU
        public int ClientId { get; set; }
}
```

Le contexte

- L’utilisation d’Entity Framework en code first nécessite de créer une classe "Manager" qui va représenter le contexte de données et faire le lien avec la base.
- La réalisation de cette classe s'effectue de la façon suivante :
  - Héritage de la classe "DbContext"
  - Définition d'une propriété de type "DbSet" pour chaque classe Métier (appelé EntitySet)
  - On définira la chaine de connexion comme en DbFirst

```
public class LocationCodeFirstContext : DbContext
{
        public DbSet<CLIENT> Clients { get; set; }
}
```

Paramétrage du nom de vos tables

- Il est possible de paramétrer le nom de vos tables, de vos colonnes qui seront générées à travers deux procédés :
  - En redéfinissant la méthode "OnModelCreating" de la classe du contexte => Code First Fluent API
  - En utilisant les annotation dans votre classe Métier

```
protected override void OnModelCreating(DbModelBuilder modelBuilder {

            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<Message>().ToTable("T_Messages", "dbo");

            modelBuilder.Entity<Message>().Property(m =>
m.Contenu).HasMaxLength(30);

            modelBuilder.Entity<Message>().Property(m =>
m.Contenu).IsRequired();

        }
```

Utilisation des annotations

- Les différentes configurations peuvent être directement effectuées sur la classe métier en utilisant les annotations

```
[Table("CLIENT")]
public class Client {
            [Key]
            [Column("ID")]
            public long Id { get; set }
            [Column("NAME")]
            [Required]
            [MaxLength(30)]
            public string Name { get; set; }
            public int PaysId { get; set; }
            public virtual Pays Pays { get; set; } }
```

Les principales annotations

- [Key] : Indique la clé primaire de la table
- [MaxLength] / [MinLength] : Taille de la chaine de caractère. Fait une verification côté client, serveur et créer un champ limité en base de données
- [StringLength] : Raccourci à MaxLenght et MinLenght
- [ConcurrencyCheck] : Marque la concurrence d'accès
- [Required] : Indique que le champ est requis
- [Column] : Défini le nom de la colonne
- [Table] : Défini le nom de la table
- [NotMapped] : Indique que le champs n'est pas mappé en base
- [ForeignKey] and [InverseProperty] : Pour les clés étrangères dans des cas spécifiques

Convention de nommage

- Pour éviter à avoir trop d'attributs à rajouter on va respecter des conventions de nommages pour qu'Entity Framework reconnaisse nos contraintes automatiquement

```
public class Department {
        public int DepartmentID { get; set; } // Primary key ou DepartmentId ou Id
        public virtual ICollection<Course> Courses { get; set; } // Navigation property
    }
    public class Course {
        public int CourseID { get; set; } // Primary key
        public int DepartmentID { get; set; } // Foreign key
        public virtual Department Department { get; set; } // Navigation properties
    }
```

https://docs.microsoft.com/fr-fr/ef/core/modeling/relationships?tabs=fluent-api%2Cfluent-api-simple-key%2Csimple-key

Update

- Pour mettre à jour la base de données on va passer par les migrations EF
- Grâce à la ligne de commande NuGet (Tools > NuGet Package Manager > Package Manager Console) on va pouvoir lancer une mise à jour de base
- A chaque changement de nos models on va créer une migration
  - Add-Migration NOM_MIGRATION
- Pour mettre à jour la base
  - Update-Database
- EF va regarder l'état de la base et passer toute les migrations qui n'ont pas encore été faites
- On pourra aussi faire un update automatique au lancement de l’application

```
context.Database.Migrate();
```

Chaine de connexion

- La méthode OnConfiguring permet de configurer le contexte avec la chaine de connexion, les logs etc…

```
 protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder) {

      if (!optionsBuilder.IsConfigured) {


      optionsBuilder.UseSqlServer(
"Server=(localdb)\\mssqllocaldb;Database=Location;Trusted_Connection=T
rue");

      }

}
```

Données par défaut

- On peut mettre des données par défaut lors de la création de la BDD

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            var u1 = new User { Id = 1, Name = "U1", Address = "test", CountryId = 1 };
            var m1 = new Message { Id = 1, Content = "test", UserId = 2 };

            modelBuilder.Entity<User>().HasData(new List<User> { u1 });
            modelBuilder.Entity<Message>().HasData(new List<Message> { m1 });

            base.OnModelCreating(modelBuilder);
        }
```

Utilisation

Module 3

Accès à votre base

- Pour accéder à votre base de données il faut passer par une instance de votre contexte
- Avec ce contexte vous pourrez accéder à vos DbSet qui représentent vos classes sur lesquels vous pourrez faire des requêtes LINQ qui seront générées en SQL
- Le contexte va se charger de traquer les objets qui sont ajoutés ou modifiés pour faire l'update à la fin
- Pour les propriétés en Lazy Loading (chargement à la demande, grâce au mot clé virtual), le contexte va retourner des dynamicProxies qui seront converti en votre type à la demande

Recherche d'un élément avec Linq

- La source de donnée à requêter sera une des entités de votre contexte

```
public CLIENT Rechercher(string emetteur)

{

          LocationCodeFirstContext context = new
LocationCodeFirstContext();

         CLIENT clt = context.Clients.FirstOrDefault(m => m.NOM ==
emetteur);

        return clt;

}
```

Insertion d’un élément

- L’insertion d’un objet se réalise à l’aide de la méthode Add de l'entité :

```
public void ajouter(CLIENT clt)
{
    LocationCodeFirstContext context = new LocationCodeFirstContext();
    context.Clients.Add(clt);
    context.SaveChanges();
}
```

- Si un client contient un autre objet, la liaison PK FK sera géré automatiquement
- La mise à jour d'entités se réalise en modifiant les propriétés souhaitées et en validant avec la méthode "SaveChanges".

Update

- Une des bonnes pratiques pour l'update est de récupérer l'objet voulu et mettre à jour les propriétés que l'on veut modifier

```
public void modifier(CLIENT paramClient)
{
    LocationCodeFirstContext context = new LocationCodeFirstContext();
    CLIENT clt = context.Clients.Find(paramClient.ID);
    clt.NOM = paramClient.NOM;
    context.SaveChanges();
}
```

Update – EF Core 7

- EF 7 permets de faire un update sans avoir besoin de récupérer l'élément d'abord
  - Fonctionne pour tous les éléments trouvés avec le where
  - Fait un SavesChanges automatiquement
  - Fonctionne avec les propriétés de navigation sans faire d'include

```
public void modifier(CLIENT paramClient) {

   LocationCodeFirstContext context = new LocationCodeFirstContext();

   var nbRows = await context.Clients.Where(c => c.Id == paramClient.ID)
.ExecuteUpdateAsync(

            updates => updates

                .SetProperty(c => c.NOM, paramClient.NOM));

}
```

Remove

- Une des bonnes pratiques pour supprimer est de récupérer l'objet voulu et de le supprimer ensuite

```
public void supprimer(int idClient)
{
    LocationCodeFirstContext context = new LocationCodeFirstContext();
    CLIENT clt = context.Clients.Find(idClient);
    context.Clients.Remove(clt);
    context.SaveChanges();
}
```

Remove – EF Core 7

- EF 7 permets de faire un remove sans avoir besoin de récupérer l'élément d'abord
  - Fonctionne pour tous les éléments trouvés avec le where
  - Fait un SavesChanges automatiquement
  - Fonctionne avec les propriétés de navigation sans faire d'include

```
public void supprimer(int idClient)
{
    LocationCodeFirstContext context = new LocationCodeFirstContext();
    var nbRows = await context.Clients.Where(c => c.Id == idClient)
.ExecuteDeleteAsync();
}
```

Aspects avancés

Module 4

Les méthodes SQL disponibles

- Attention, dans une requête LINQ, si vous mettez une méthode qui n'est pas traduisible en SQL alors la requête ne va pas fonctionner

```
CLIENT clt = context.Clients.FirstOrDefault(m =>
IsStringEqualsCustom(m.NOM, emetteur));
```

- Certaines méthodes sont traduisible

```
CLIENT clt = context.Clients.FirstOrDefault(m =>
m.NOM.ToLower().Contains(emetteur.ToLower()));
```

- Une liste est disponible dans EF.Functions

```
context.Clients.Where(c => EF.Functions.Like(c.Nom, "%D__M%"));
```

Le tracking

- EF traque les objets récupérés de la base de données pour détecter des changements pour pouvoir ensuite faire les mis à jour
- Lorsque l'on récupère nos données pour de la lecture seule on peut désactiver ce tracking pour améliorer les performances

```
context.ChangeTracker.QueryTrackingBehavior =
Microsoft.EntityFrameworkCore.QueryTrackingBehavior.NoTracking;
```

- On peut aussi supprimer le tracking pour juste une méthode

```
CLIENT clt = context.Clients.AsNoTracking().FirstOrDefault(m => m.NOM ==
emetteur);
```

- Comme LINQ, certaines méthodes sont des méthodes d'extensions du namespace Microsoft.EntityFrameworkCore, il ne faut donc pas oublier le using pour les avoir

Requête SQL brute

- On peut lancer nos propres requêtes SQL

```
var clients = context.Client.FromSqlRaw("SELECT * FROM
CLIENT").OrderBy(c => c.Id).ToList();

CLIENT client = context.Client.FromSqlRaw("EXECUTE GetClientById " +
id);

MyClass c = context.Set<MyClass>().FromSqlRaw("SELECT XXX");
```

- Pour récupérer des objets fortement typés il faut que les noms et les types des champs retournés dans la requête SQL soient les mêmes que les propriétés de nos classes

Requête asynchrone

- Pour améliorer les performances de l'application on peut lancer nos requêtes LINQ en asynchrone

```
var clients = await context.Client.OrderBy(c => c.Id).ToListAsync();
```

- Dans le même principe on peut faire de la sauvegarde asynchrone avec la méthode SaveChangesAsync

Les transactions

- On peut démarrer des transactions avec EF, sous réserve que le SGBDR le supporte

```
using (var transaction = context.Database.BeginTransaction()) {
try
{
        ...
        transaction.Commit();
}
catch (Exception e)
{
        transaction.Rollback();
}
```

Les logs

- On peut facilement voir les logs de EF pour voir les requêtes générées

```
optionsBuilder.LogTo(Console.WriteLine);
```

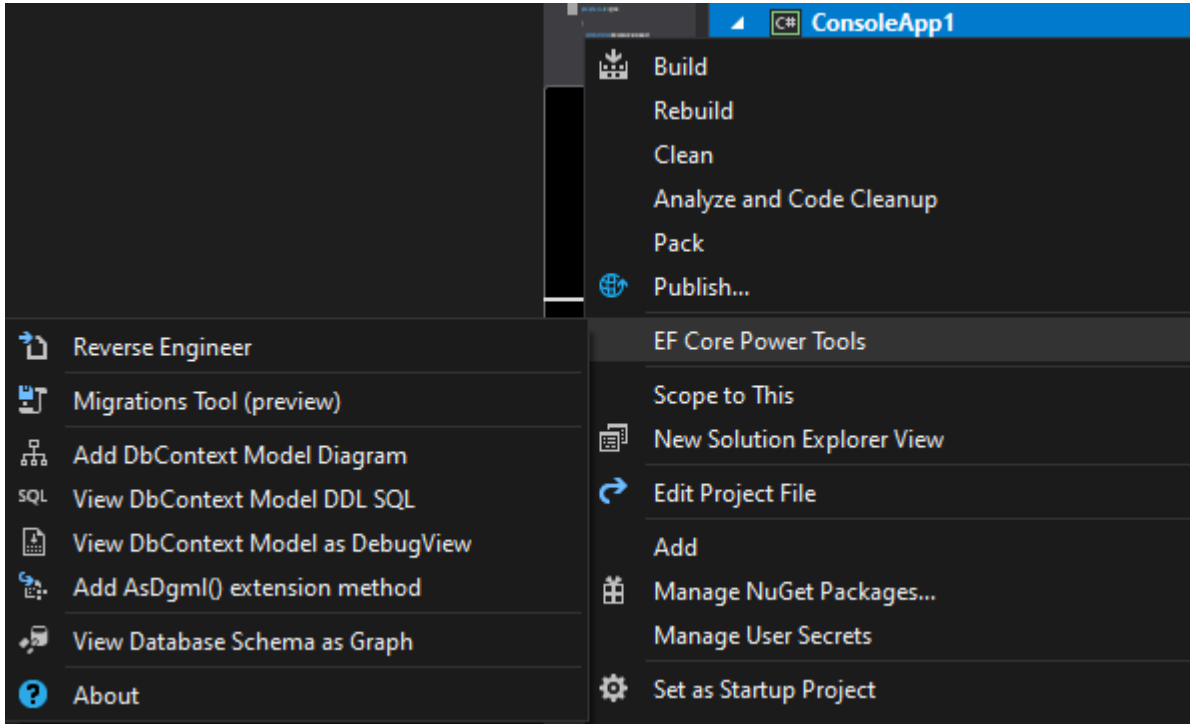
- En version 5 on pourra aussi le voir directement en point d'arrêt ou avec la méthode `ToQueryString`

EF Core Power Tools

Module 5

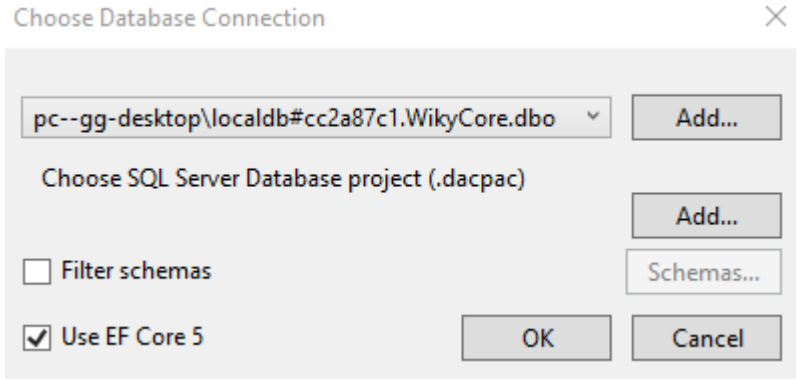
EF Core Power Tools

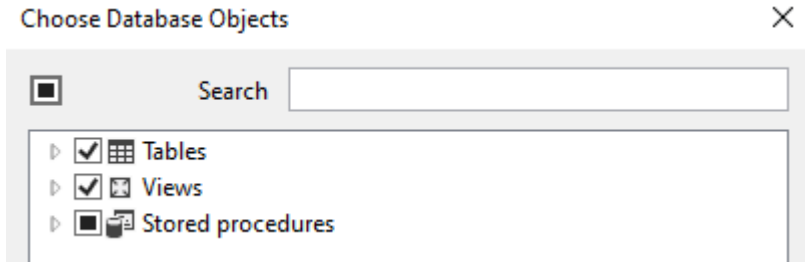
- EF Core Power Tools est un outil pour simplifier l'utilisation d'EF Core, surtout en mode DbFirst
- Ce plugin Visual Studio va nous permettre d'avoir un menu contextuel en plus sur nos projets afin d'utiliser EF Core en IHM

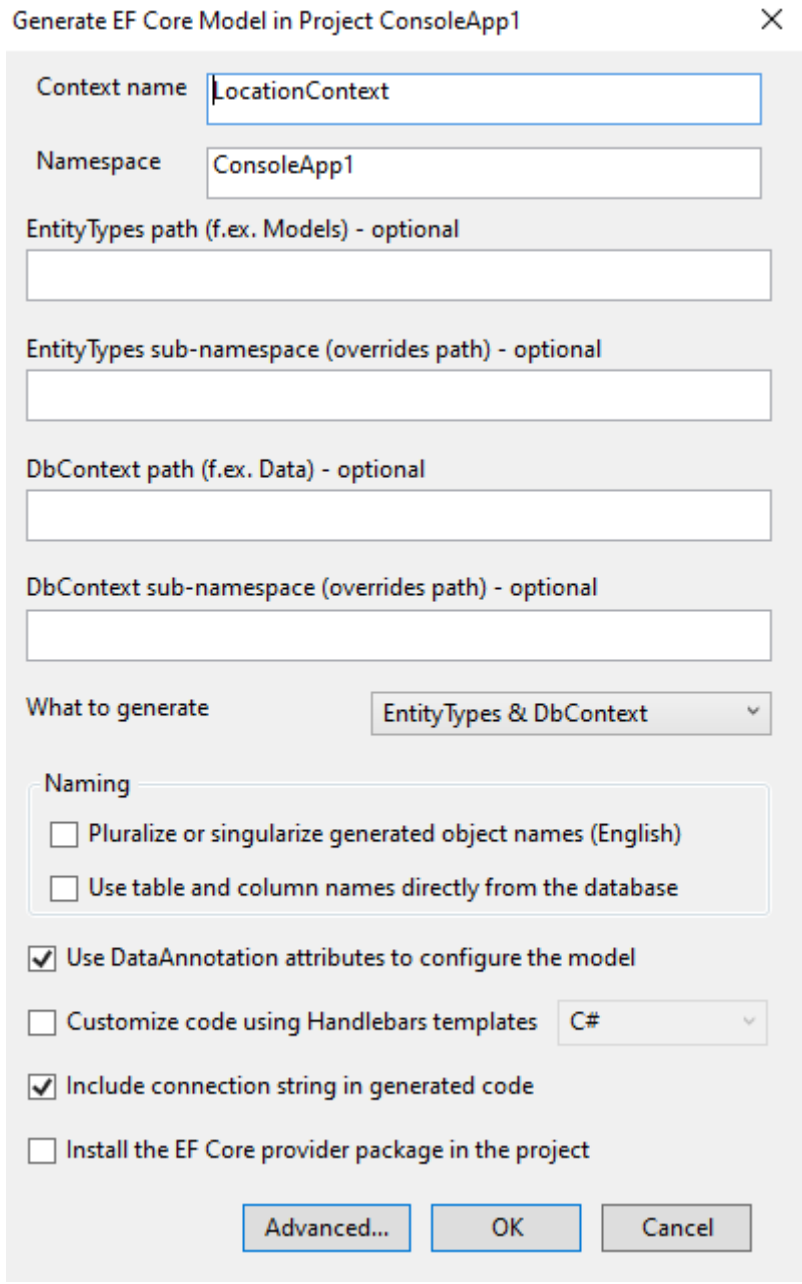


Reverse Engineer

- Permet de faire la génération de code avec un DbFirst

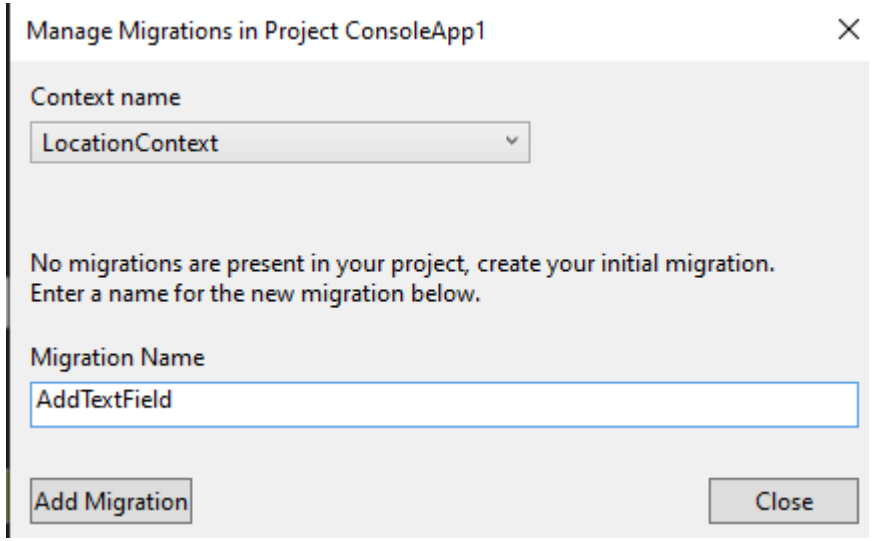


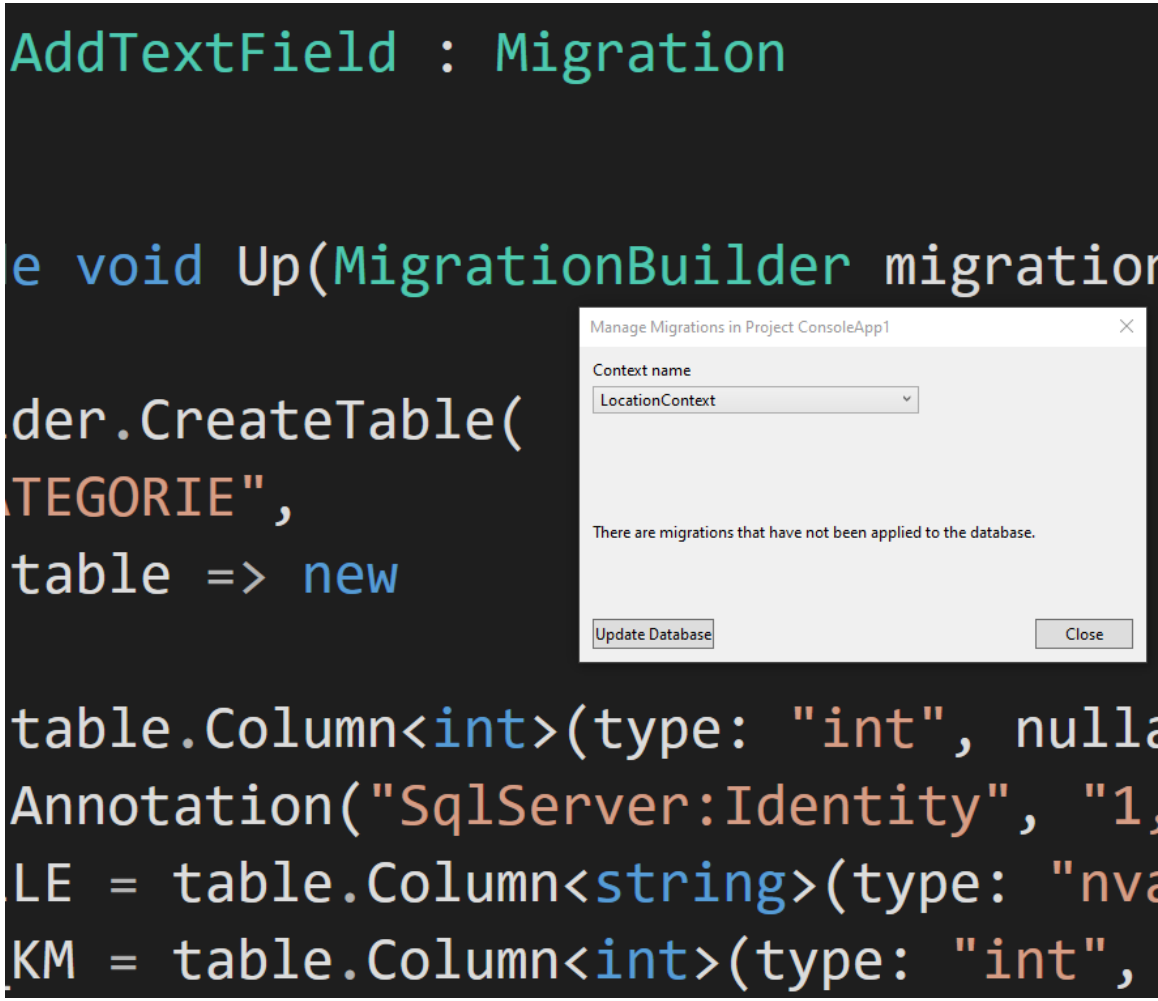




Migrations Tool

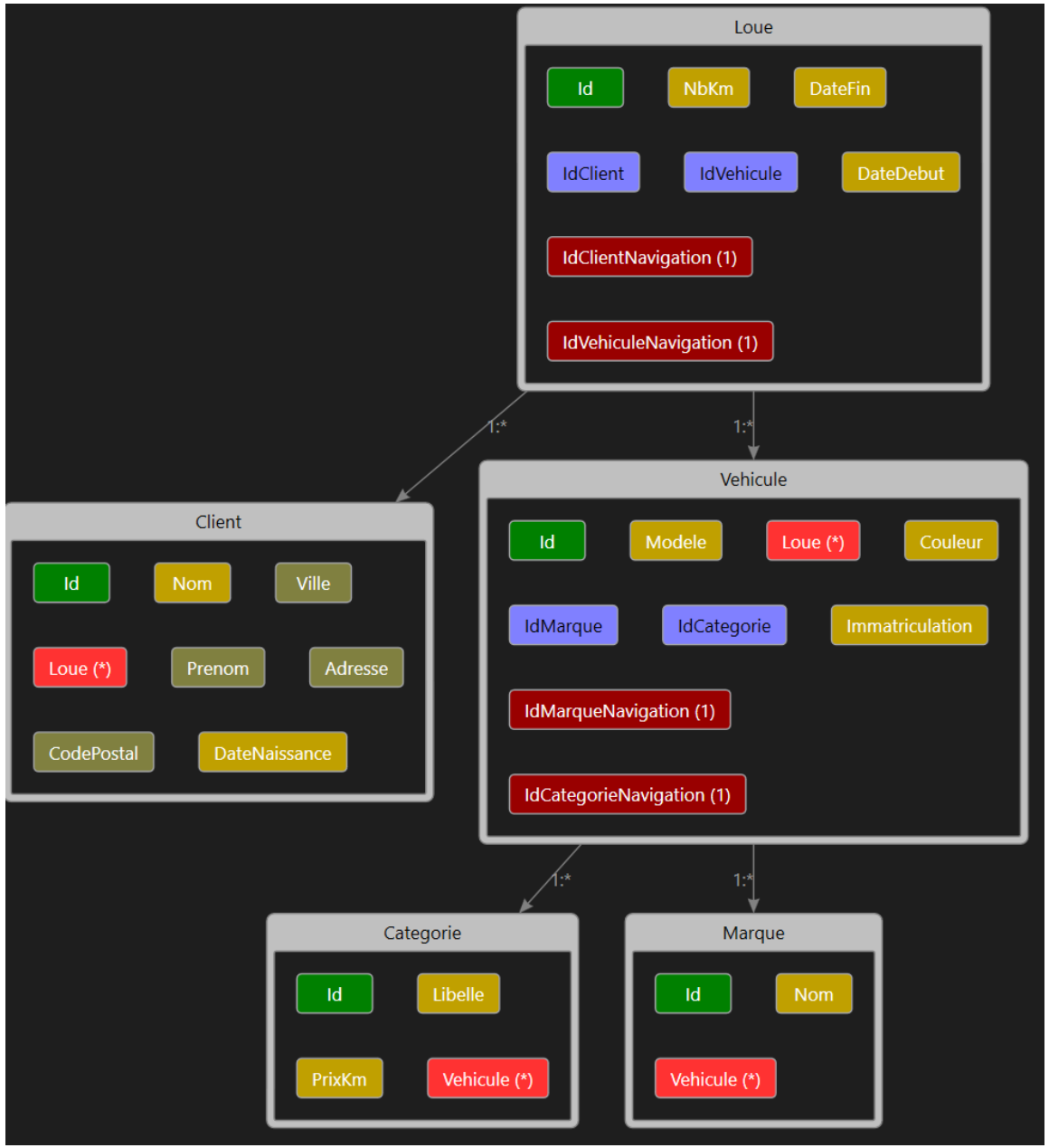
- Permet de gérer les migrations en CodeFirst





Add DbContext Model Diagram

- Permet de créer un diagramme à partir de la base



View DbContext Model DDL SQL & as DebugView

- DDL SQL
  - Permet de récupérer le SQL de création de base

```
CREATE TABLE [MARQUE] (
    [ID] int NOT NULL IDENTITY,
    [NOM] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_MARQUE] PRIMARY KEY ([ID])
);
```

- DebugView
  - Permet de récupérer la configuration détectée par EF Core

```
Model:
  EntityType: Categorie
    Properties:
      Id (int) Required PK AfterSave:Throw ValueGenerated.OnAdd
        Annotations:
          Relational:ColumnName: ID
```

Fin de la formation